

JS Level 1



Введение



DOM API

Эта лекция будет сложной, поскольку мир DOM API во многих местах неоднозначный и в нём очень много понятий и сущностей. Мы не рассчитываем на то, что вы с первого прочтения поймёте всё то, что описано в первой части лекции. Но понять это нужно.

Ключевое, как и всегда, в том, что не все они используются, и, самое главное, – понимать основные идеи и запоминать правила (и исключения из них).

Это больше теоретическая лекция, нам нужно, чтобы вы понимали, откуда и что берётся. Поэтому сделайте следующее упражнение: когда читаете лекцию, пишите на листе бумаги понятия и связывайте их стрелками, чтобы понять взаимосвязь.



DOM API

Важно даже не столько понимание самой лекции, сколько общее представление о том, как всё устроено. Потому что на текущий момент вы, скорее всего, не поймёте всех нюансов (обучение программированию итеративно: сначала вы понимаете общую концепцию, затем детали реализации, затем специфические нюансы).



DOM API

При этом крайне важно стараться понять логику внутреннего устройства именно с точки зрения вашего становления как разработчика, потому что без этого вы будете, что называется "monkey programmer" – это программист, который не понимая причин происходящего, наугад пробует любые решения, которые он где-то скопипастил, ему подсказали и т.д. И на собеседовании вы хотя бы будете знать, что такое спецификация и откуда берутся те объекты, свойства и методы, которые вы используете.

Старайтесь понять причины и общую схему. Если вы поймёте это – в вашей голове выстроится структура, на базе которой вы сможете строить всё своё дальнейшее обучение.



Повторение



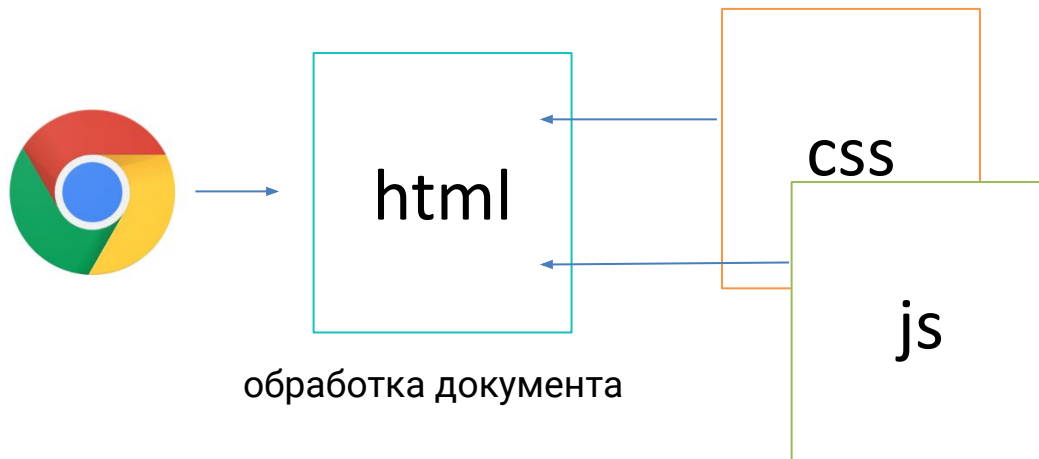
Web Application

На прошлой лекции мы поговорили с вами, как работают веб-приложения (а именно их клиентская часть) – они загружаются и запускаются в браузере:



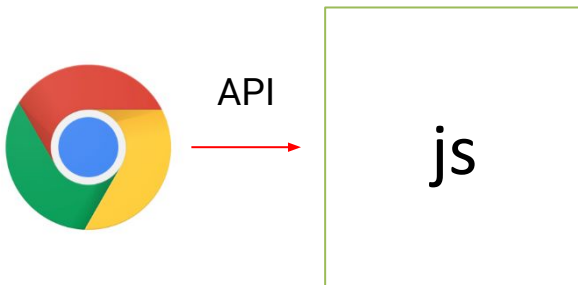
Ресурсы

Кроме того, мы обсудили сам механизм: сначала загружается HTML-документ (если вы указали его в адресной строке), а затем уже все ресурсы, которые в этом самом документе прописаны:



API

После обработки документа браузер предоставляет нам API (набор готовых объектов, которые мы можем использовать из нашей программы), в том числе для управления страницей:



Вы можете спросить, зачем нам вообще это API и DOM API, в частности? Ответ достаточно простой: без API мы бы не могли управлять страницей из JS, а это значит, что сам JS стал бы не особо нужен.



DOM API



Задача

Мы хотим сделать так, чтобы при загрузке наша страничка выглядит вот так:



Hello, JS!

А через 2 секунды, мы хотим её поменять через JS, чтобы она выглядела вот так:



Hello, Alif Skills!



Изображения

Изображения мы взяли по адресу:

https://alif-skills.pro/media/logo_js.svg

https://alif-skills.pro/media/logo_alif.svg

И сохранили в каталоге `img`:

```
> css
  ✓ img
    | logo_alif.svg
    | logo_js.svg
  > js
  <> index.html
```



Задача

Начнём пока без картинок с вот такой простой структуры:

```
<> index.html > ...  
1  <!DOCTYPE html>  
2  <html lang="en">  
3  <head>  
4      <meta charset="UTF-8">  
5      <meta name="viewport" content="width=device-width, initial-scale=1.0">  
6      <title>Document</title>  
7      <link rel="stylesheet" href="css/styles.css">  
8  </head>  
9  <body>  
10     <h1>Hello, JS!</h1>  
11     <script src="js/app.js"></script>  
12 </body>  
13 </html>
```



DOM

DOM описан в спецификации (<https://dom.spec.whatwg.org/>), но разобраться в ней без подготовки достаточно сложно.

Мы начнём по-простому, и скажем следующее: DOM API нам предоставляет готовый объект с именем `document`, который и является "окном" в мир загруженного документа.

Примечание*: вы периодически будете встречать термины DOM и DOM API. В большинстве случаев, когда говорят DOM имеют в виду либо DOM API (т.е. предоставляемый браузером набор готовых объектов и функций), либо DOM-дерево (т.е. созданную браузером из вашей разметки иерархию родитель-дети из объектов).



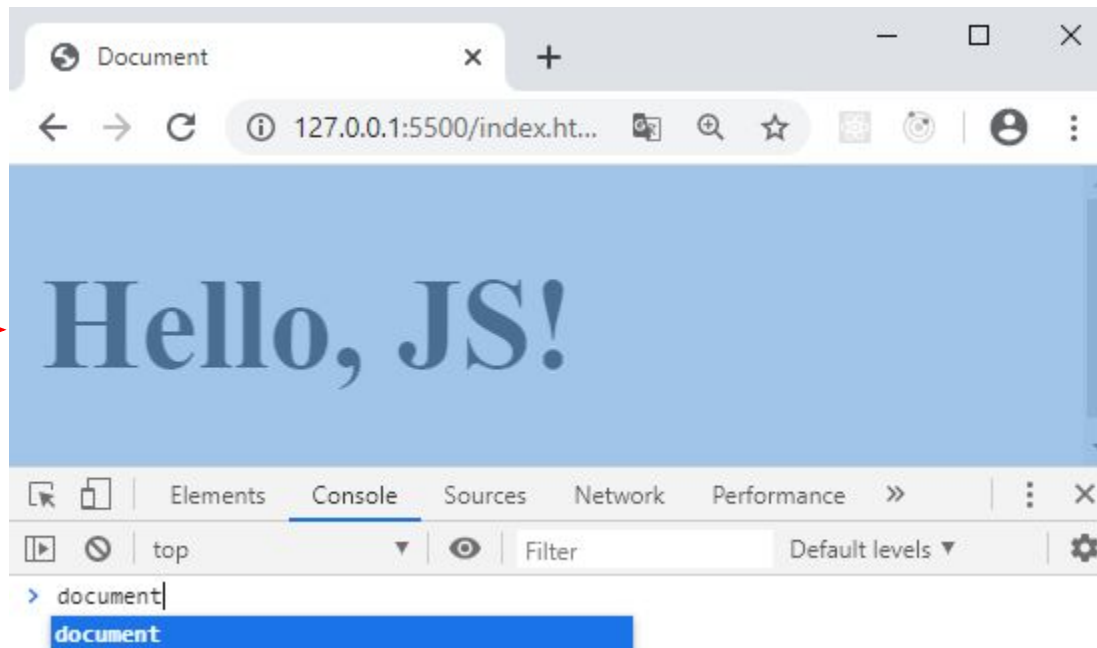
DOM

Мы в рамках нашей лекции под DOM будем иметь в виду именно DOM API. А когда будет идти речь об объектах, организованных в иерархию, будем говорить DOM-дерево.



document

Давайте посмотрим на него, для этого в консоли начнём набирать `document`:



Обратите внимание, как только вы начинаете в консоли набирать какой-то объект, который имеет визуальное представление в DOM (т.е. его видно и у него размеры больше чем `0px` на `0px`), то он вам сразу подсвечивается на страничке.



DOM

Если наберём до конца и раскроем объект (как это делали раньше), то увидим достаточно удивительную вещь: отображаются не свойства, а разметка:

```
> document
< ▼#document
  <!doctype html>
  <html lang="en">
    ▶ <head>...</head>
    ▶ <body>...</body>
  </html>
```

Это одно из "хитрых" поведений консоли: если она считает, что какой-то объект принадлежит к миру DOM, то она печатает его представление именно таким образом.

Также будет выводить объекты и `console.log()`, если вы осуществляете вызов из скрипта (`js/app.js`).



DOM

Для того, чтобы вывести что-то, принадлежащее к миру DOM как объект, нужно использовать `dir` (либо `console.dir` в скрипте):

```
> dir(document)
```

```
▼ #document 1
  URL: "http://127.0.0.1:5500/index.html"
  ▶ activeElement: body
  ▶ adoptedStyleSheets: []
  alinkColor: ""
  ▶ all: HTMLAllCollection(8) [html, head, meta, meta, title, body, h1, ...]
  ▶ anchors: HTMLCollection []
  ▶ applets: HTMLCollection []
  baseURI: "http://127.0.0.1:5500/index.html"
  bgColor: ""
  ▶ body: body
    characterSet: "UTF-8"
    charset: "UTF-8"
    childElementCount: 1
  ▶ childNodes: NodeList(2) [html, html]
  ...
  ▶ __proto__: HTMLDocument
```

Мы приводим не все свойства для краткости

ВОТ СТОЛЬКО СВОЙСТВ
БУДЕТ У ОБЪЕКТА

Конечно, количество свойств впечатляет, и вряд ли вы их все выучите в первое время. Поэтому нужно научиться пользоваться документацией, чтобы понимать, что и откуда берётся.



Скрипт

Термином "скрипт" обозначают то, что вы подключаете с помощью тега `script`. Т.е. в нашем случае – это всё наше приложение в виде JS-файла.



HTMLDocument

Первое, что нужно запомнить: искать нужно не `document`, а то, что написано в `__proto__` - ищем в Google: [MDN HTMLDocument](#)

► `__proto__`: HTMLDocument

В новых версиях будет отображаться как:

► `[[Prototype]]`: HTMLDocument

HTMLDocument - это абстрактный интерфейс `DOM`, который обеспечивает доступ к специальным свойствам и методам, не представленным по умолчанию в регулярном (XML) документе.

Его методы и свойства включены в страницу `document` и перечислены отдельно в их собственном разделе на вышеупомянутой связанной странице DOM.

Статья из MDN
(лучше читать на
английском)

Разбираем по словам: абстрактный интерфейс. Слово абстрактный можно убрать, а вот слово интерфейс интересное. Что это такое?



HTMLDocument

Помните, мы говорили про API (Application Programming Interface)? Тогда это был набор объектов, а вот этот ([HTMLDocument](#)) – это описание конкретных свойств и методов, которые должны быть у объекта, который нам предоставляют.



Document

кликаем сюда



Его методы и свойства включены в страницу `document` и перечислены отдельно в их собственном разделе на вышеупомянутой связанной странице DOM.

На открывшейся странице будет доступно много всякой информации, но нас будут интересовать всего два момента (ну и свойства, конечно же, полистайте):

Чаще всего используется прямой доступ к объекту **document** из сценариев `scripts` которые подгружаются документом. (Этот же объект доступен как `window.document`.)



Примечание: Интерфейс `Document` наследует также интерфейсы `Node` и `EventTarget`.

Давайте разбираться.



Глобальный объект

Global Object



Web API

DOM API – это лишь часть того API, которое предоставляет нам браузер. Например, объект `console`, который мы с вами использовали, никак к DOM API не относится.

В JS существует понятие глобального объекта – это специальный объект, в котором ищутся все имена, не найденные в текущем блоке. Что это значит? Давайте напишем `app.js` (подключим к html как обычно):

```
js > JS app.js  
1 console.log('worked!');
```

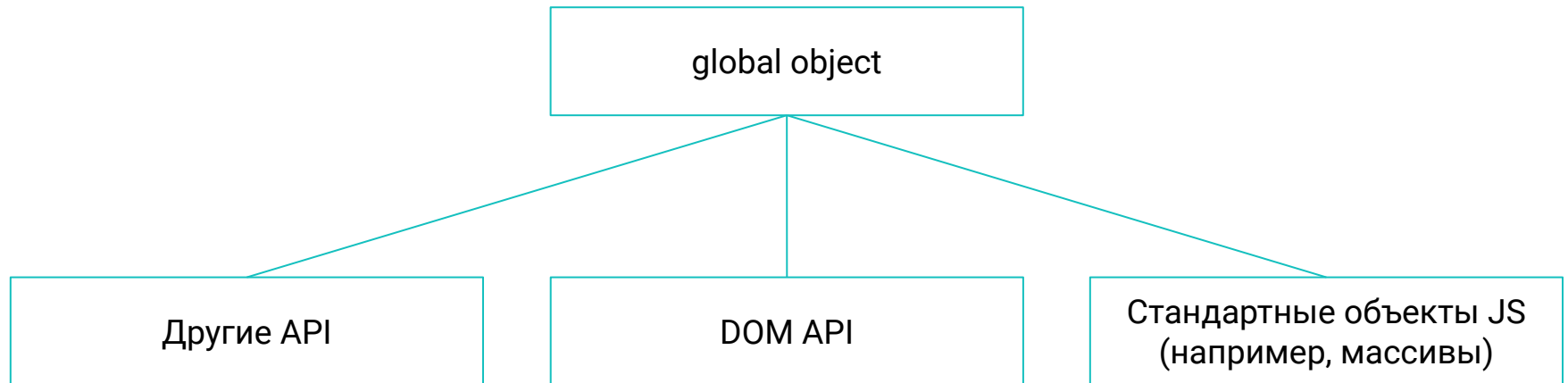
Откуда берётся объект `console`? Мы же не объявляли такое имя. На самом деле, если в нашем коде это имя не объявлено, то браузер спереди этого имени дописывает `globalThis` (работает в новых браузерах, в более старых – `window`):

```
js > JS app.js  
1 globalThis.console.log('worked!');
```



global object

Условно, на картинке вы можете представить это себе следующим образом:



globalThis

Давайте попробуем распечатать этот объект:

```
> globalThis
```

```
< ▼ Window {parent: Window, opener: null, top: Window, length: 0, frames: Window, ...} ⓘ
```

```
  ▶ GetParams: f (t)
```

```
  ▶ alert: f alert()
```

```
  ▶ applicationCache: ApplicationCache {status: 0, oncached: null, onchecking: null, ondownloading: null, onerror: null, ...}
```

```
  ▶ atob: f atob()
```

```
  ▶ blur: f blur()
```

```
  ▶ btoa: f btoa()
```

```
  ▶ caches: CacheStorage {}
```

```
  ▶ cancelAnimationFrame: f cancelAnimationFrame()
```

```
  ▶ cancelIdleCallback: f cancelIdleCallback()
```

```
  ▶ captureEvents: f captureEvents()
```

```
  ▶ chrome: {loadTimes: f, csi: f}
```

```
  ▶ clearInterval: f clearInterval()
```

```
  ▶ clearTimeout: f clearTimeout()
```

```
...
```

```
  ▶ document: document
```

```
  ▶ external: External {}
```

```
  ▶ fetch: f fetch()
```

```
...
```

```
  ▶ history: History {length: 2, scrollRestoration: "auto", state: null}
```

```
  ▶ indexedDB: IDBFactory {}
```

```
    innerHeight: 150
```

```
    innerWidth: 1366
```



globalThis

Обратите внимание, там действительно есть и `alert`, и `document`.

Если вы не совсем внимательно просматривали вывод, то можете обвинить нас в том, что мы вас обманываем и `console`-то, как раз там нет. Всё дело в том, что не все свойства объектов выводятся в алфавитном порядке, поэтому вам придётся нажать `Ctrl + F`, чтобы найти `console`:

```
▼ console: console
  ▶ assert: f assert()
  ▶ clear: f clear()
  ▶ context: f context()
  ▶ count: f count()
  ▶ countReset: f countReset()
  ▶ debug: f debug()
  ▶ dir: f dir()
  ▶ dirxml: f dirxml()
  ▶ error: f error()
  ▶ group: f group()
  ▶ groupCollapsed: f groupCollapsed()
  ▶ groupEnd: f groupEnd()
  ▶ info: f info()
  ▶ log: f log()
```



globalThis

И самое, смешное, что внутри объекта `globalThis`, есть свойство `globalThis`:

```
▼ globalThis: Window  
  ► GetParams: f (t)  
  ► alert: f alert()
```

Это то же самое, что вы в своём телефоне бы записали свой номер телефона.



globalThis

Ключевое: любые имена, которые вы используете (без объявления переменных или функций), ищутся как свойства глобального объекта.

И самое плохое в этом следующее, если вы случайно не напишите `let` или `const`, то создадите свойство в глобальном объекте:

```
js > JS app.js  
1  amount = 50;
```

В `globalThis` появится свойство `amount`.



Strict Mode

Чтобы запретить такое поведение, внедрили специальную директиву ("строку" в специальном формате), которая позволяет "ужесточить" требования к JS:

```
js > JS app.js  
1  'use strict';  
2  
3  amount = 50;
```

← пишите первой строкой в файле

Теперь этот код будет генерировать ошибку:

```
✖ ▶ Uncaught ReferenceError: amount is not defined  
   at app.js:3
```

>

Начиная с сегодняшней лекции бот будет требовать наличия этой директивы во всех ваших скриптах.



Замечание

Вам может показаться неинтересным изучать все эти нюансы и т.д. Но здесь важно научить себя и свой мозг тому, чтобы разобраться в деталях и воспринимать такие особенности как своего рода забавные головоломки.



Window

Давайте поговорим немного о `Window` (или `globalThis` в современных браузерах) Давным-давно у браузеров не было вкладок (можете загуглить IE5, IE6) и каждый сайт открывался в новом окне браузера. Так вот объект `window`, предоставлял как раз-таки доступ к этому окну (можно было делать всякие нехорошие сайты, которые из одного окна открывают кучу других – сейчас браузеры блокируют эту возможность), но в консоли вы можете написать (откроется новая вкладка):

```
> open('https://google.com')
```

Обратите внимание, мы написали `window` во второй раз с маленькой буквы, а не с большой. Почему? Потому что имя `globalThis` появилось только в стандарте 20-го года, до этого в браузере глобальный объект был известен под именем `window` (и в старых статьях вы будете встречать его именно под этим именем). Осталось разобраться только, что такое `Window`.



Window

И тут мы переходим ко второму способу работы с документацией, это чтение спецификации. Объект [Window](#) описан в спецификации [HTML](#) следующим образом:

```
IDL [Global=Window,
     Exposed=Window,
     LegacyUnenumerableNamedProperties]
interface Window : EventTarget {
    // the current browsing context
    [LegacyUnforgeable] readonly attribute WindowProxy window;
    [Replaceable] readonly attribute WindowProxy self;
    [LegacyUnforgeable] readonly attribute Document document;
```



Window

Вам, скорее всего, будет ничего не понятно. Но мы попробуем разобраться.

Во-первых, всё, что тут написано, к JS никакого отношения не имеет. Почему? Потому что когда это всё создавалось, было решено сделать всё нейтральным по отношению к языку программирования (а вдруг в вебе будет не только JS). В итоге эти ребята (авторы) придумали ещё один язык, задача которого, описывать объекты предоставляемые нам API.

Может показаться не совсем логичным, но да, существуют языки, задача которых просто описывать объекты и ни для чего иного не использоваться.



WEB IDL

Этот язык называется [WEB IDL](#) (Interface Description Language) и на самом деле, всё не так уж сложно:

```
IDL [Global=Window,  
    Exposed=Window,  
    LegacyUnenumerableNamedProperties]  
interface Window : EventTarget {
```

- **Global** – объект может использоваться в качестве глобального
- **Exposed** – объект доступен в контексте окна браузера (т.е. в том режиме, в котором мы работаем)
- **:** – этот объект включает свойства и методы, которые описаны в другом интерфейсе (**EventTarget** в нашем случае)



WEB IDL

```
[LegacyUnforgeable] readonly attribute Document document;  
attribute DOMString name;
```

- **readonly attribute** – попытка записать в это свойство ничего не даст (и ошибки тоже)
- **DOMString** – обычный тип **string** в JS

```
WindowProxy? open(optional USVString url = "", optional DOMString target =  
"_blank", optional [LegacyNullToEmptyString] DOMString features = "");
```

- **?** – значит, что может вернуть **null** (про **null** чуть позже)
- **optional** – необязательно указывать (опционально)
- **= ""** или **= "_blank"** – значение по умолчанию, если вы ничего не передали

```
void alert(DOMString message);
```

- **void** – функция ничего не возвращает (**undefined** в JS)



WEB IDL

```
Window includes GlobalEventHandlers;  
Window includes WindowEventHandlers;
```

- **includes** – авторы поленились и сказали вам: "сходи в [GlobalHandlers](#) и сам скопируй всё сюда" (т.е. включено всё, что написано в [GlobalHandlers](#)).

Вот этого нам, в принципе, достаточно для работы. Тем более большинство свойств и методов кликабельно, вы зайти в них и почитать описание (при необходимости воспользовавшись Google Translate для непонятных фраз).



Интерфейсы

Итого: интерфейс это просто требования к тому, какие свойства (в том числе методы) должны быть у объектов.

Например, в реальной жизни: у Банка (интерфейс) должна быть лицензия (свойство) на оказание банковских услуг.



Интерфейсы

Про интерфейсы нужно запомнить две вещи:

1. Если в названии интерфейса есть `:` это значит, что этот интерфейс содержит всё то, что объявлено в интерфейсе, написанном через `:`

```
interface Window : EventTarget {
```

В наших обычных примерах это выглядело бы как `Банк : Организация` (т.е. у Банка должны быть все те же свойства, что и у обычной организации, но может быть и ряд дополнительных)

2. Если после описания интерфейса фигурирует слово `includes` значит этот интерфейс включает в себя ещё свойства и методы другого интерфейса:

```
Window includes GlobalEventHandlers;  
Window includes WindowEventHandlers;
```

Считайте, как будто бы их просто скопировали оттуда и вставили в этот интерфейс.



DOM Tree



DOM Tree

Итак, мы с вами кратко познакомились с тем, как всё устроено. Давайте, наконец, попробуем уже написать какой-то код, который будет модифицировать наше DOM-дерево.

Напоминаем, что DOM-дерево – это иерархическое (родитель-дети) представление объектов, которое получается после обработки нашей HTML-страницы браузером:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>
<body>
  <h1>Hello, JS!</h1> ←
  <script src="js/app.js"></script>
</body>
</html>
```

Для простоты будем менять текст в нашем теге **h1**.



body

Первое, что нас будет интересовать – это возможность добраться до `body` (т.к. зачем нам модифицировать `head`*)?

Для этого в объекте `document` есть свойство `body`, в котором и содержится элемент `body`:

```
> document.body  
< ▶ <body>...</body>
```

Надеемся, что вы ещё помните, почему выводится всё именно в таком виде.

Примечание*: на базе модификации элементов, содержащихся в `head` можно организовать смену иконки – `favicon` на вкладке или изменение текста на ней (например, что вам пришло новое сообщение).



body

У объекта `body` есть следующие свойства (конечно же их гораздо больше):

```
► childNodes: NodeList(9) [text, h1, text, script, text, comment, text, script, text]  
► children: HTMLCollection(3) [h1, script, script]
```

Если внимательно приглядеться, то это нечто, похожее на массив, но там написано не `Array`, а `NodeList` и `HTMLCollection` соответственно, и часть их содержимого пересекается (в обоих есть `h1` и `script`).



body

На самом деле, дерево – это иерархия (родитель и дети), так вот и то, и другое – это дети. Но давайте разбираться, чем они отличаются:

```
▼ childNodes: NodeList(9)
  ▶ 0: text
  ▶ 1: h1
  ▶ 2: text
  ▶ 3: script
  ▶ 4: text
  ▶ 5: comment
  ▶ 6: text
  ▶ 7: script
  ▶ 8: text
    length: 9
  ▶ __proto__: NodeList

▼ children: HTMLCollection(3)
  ▶ 0: h1
  ▶ 1: script
  ▶ 2: script
    length: 3
  ▶ __proto__: HTMLCollection
```



node

Посмотрим на первый элемент в списке `childNodes`:

```
▼ childNodes: NodeList(9)
  ▼ 0: text
    assignedSlot: null
    baseURI: "http://127.0.0.1:5500/index.html"
    ▶ childNodes: NodeList []
    data: "␣"
    firstChild: null
    isConnected: true
    lastChild: null
    length: 5
    ▶ nextElementSibling: h1
    ▶ nextSibling: h1
    nodeName: "#text"
    nodeType: 3
    nodeValue: "␣"
    ▶ ownerDocument: document
    ▶ parentElement: body
    ▶ parentNode: body
    previousElementSibling: null
    previousSibling: null
    textContent: "␣"
    wholeText: "␣"
    ▶ __proto__: Text
```

Свойство `textContent` представляет собой содержимое конкретной ноды (узла дерева) в виде строки. Сейчас там хранится какая-то непонятная штука, которой в нашем документе нет и не должно быть.

На самом деле, всё, что мы пишет в документе, преобразуется в ноды, и то, что мы видим – это перенос строки + отступ:

```
8 <body> ← перенос строки
9   <h1>Hello, JS!</h1>
10  <script src="js/app.js"></script>
11 </body>
12 </html>
```

отступ



Node vs Value

В `children` (`HTMLCollection`) никаких таких элементов нет. Почему?

```
▼ children: HTMLCollection(3)  
  ▼ 0: h1  
    ...  
    tagName: "H1"  
    textContent: "Hello, JS!"  
    title: ""  
    translate: true  
  ► __proto__: HTMLHeadingElement
```

`Node` – это узел в DOM-дереве, а `Element` – это элемент (например, `HTMLElement` – т.е. то, во что после парсинга превращается тег с атрибутами и содержимым).

Таким образом, если мы хотим работать с элементами (а в 99% случаев мы хотим работать именно с ними, то нам нужны элементы, а не ноды).



textContent

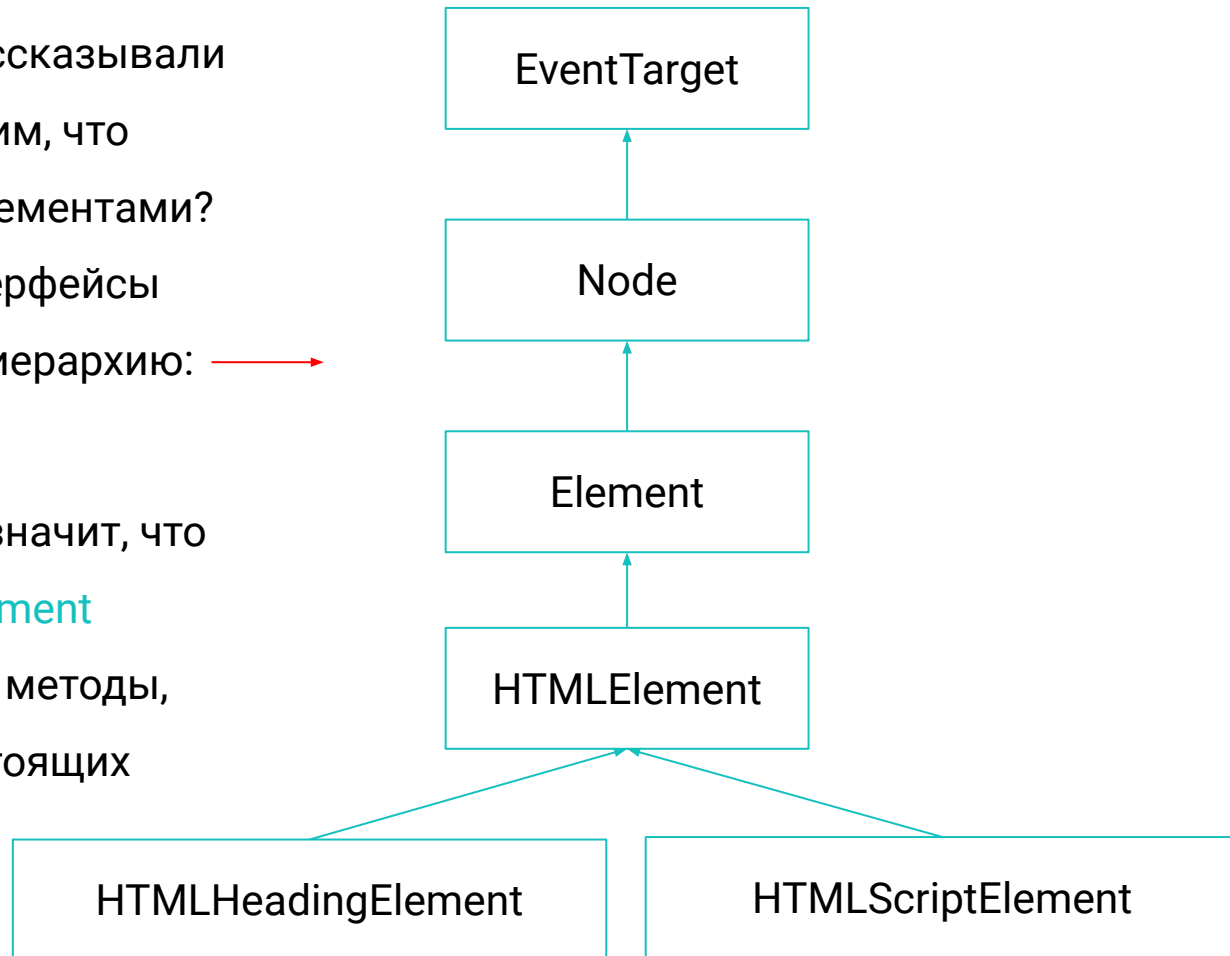
Свойство `textContent` содержит внутри себя текст, который размещён внутри ноды. В случае элементов – это тот текст, который видит пользователь, когда смотрит на элемент.



Иерархия

Так зачем мы тогда вам рассказывали про **Node**? Если сами говорим, что будем работать только с элементами? Всё дело в том, что эти интерфейсы организованы вот в такую иерархию: →

Что значит иерархию? Это значит, что интерфейс **HTMLHeadingElement** включает в себя все поля и методы, которые хранятся в вышестоящих интерфейсах (вспомните Web IDL).



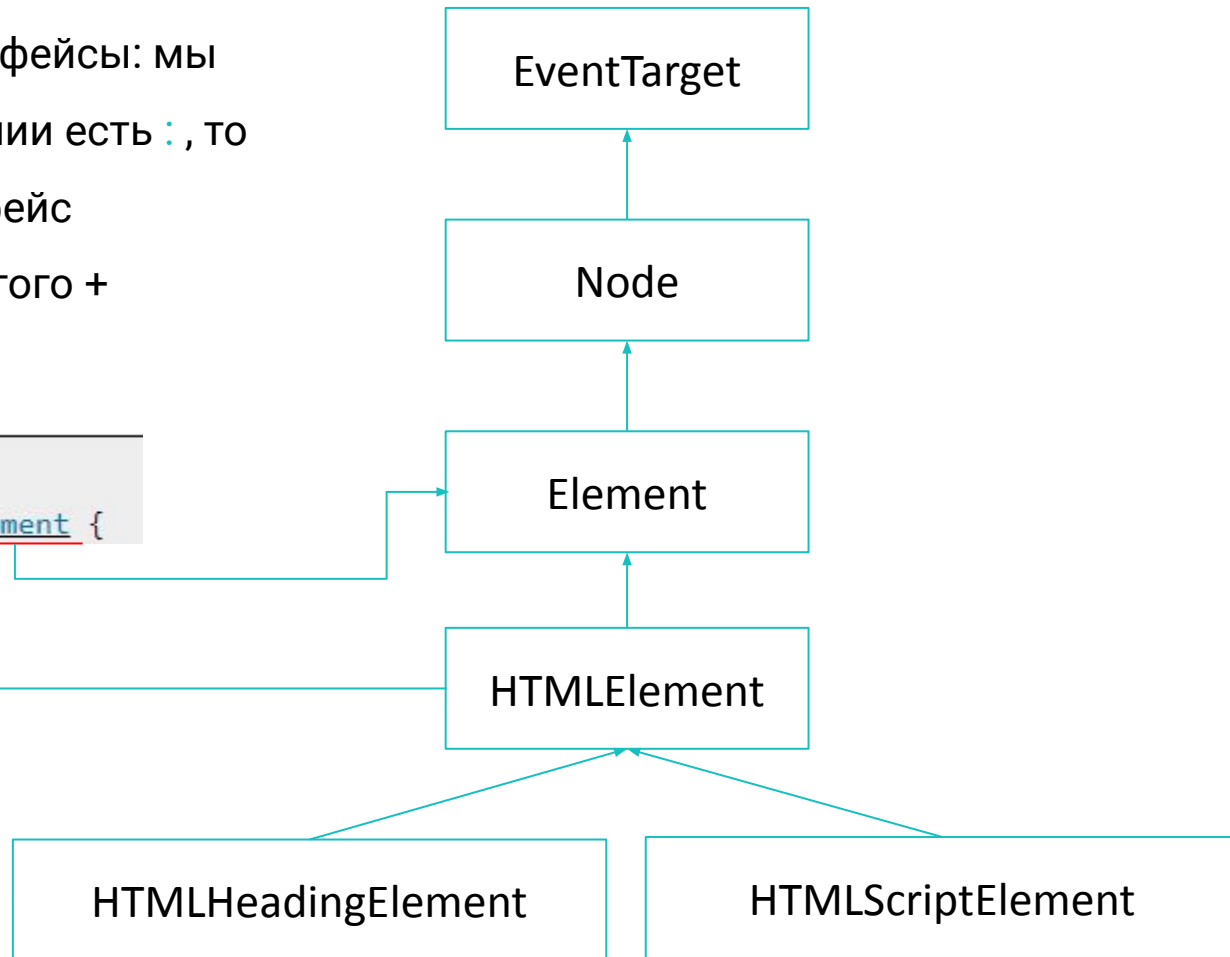
Для каждого html-тега свой интерфейс



Иерархия

Вспомните слайд про интерфейсы: мы говорили, что если в названии есть `:`, то это значит, что один интерфейс содержит все свойства другого + некоторые свои:

```
IDL [Exposed=Window]  
interface HTMLElement : Element {
```



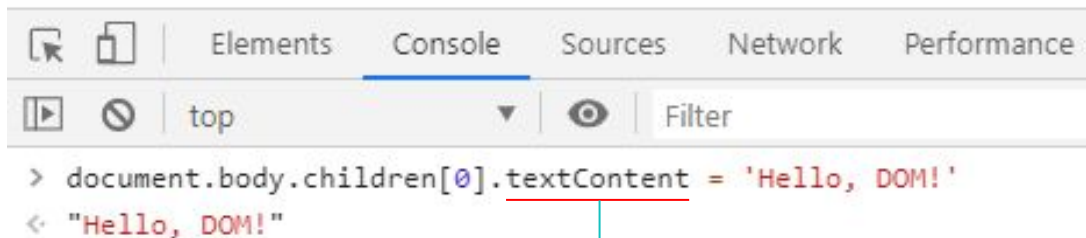
Доступ к элементу

Собираем всё вместе: мы можем добраться до элемента через `body`, в котором есть свойство `children`, в котором в нумерованном свойстве `0` хранится элемент `h1`. А в его свойстве `textContent` хранится текстовое содержимое:

```
> document.body.children[0].textContent  
< "Hello, JS!"
```

Пробуем записать:

Hello, DOM!



Работает, но как-то неудобно. А что, если там добавится ещё один элемент, тогда `h1` станет не 0-ым. Или вообще элементов будет сто, как мы запомним индекс?



Поиск элементов



Поиск элементов

Вместо того, чтобы "мучаться" с индексами, мы можем поступить так же, как и с массивами в прошлой лекции: поискать удобный метод, который будет решать задачу за нас.

И, действительно, такие методы есть прямо в `document` и называются они `querySelector` и `querySelectorAll`.

В чём суть: эти методы, позволяют нам искать элементы прямо в документе:

- `querySelector` первый найденный в DOM объект (сверху вниз), либо `null`
- `querySelectorAll` возвращает все найденные в виде списка* (список может быть пустым)

Примечание*: важно - именно в виде списка, а не массива.



null

Про `null` стоит поговорить ещё раз отдельно: `null` это специальное значение, которое показывает намеренную указание на отсутствия объекта.

Вот здесь важно различать: `undefined` – это "не назначено" ("не знаю" по-нашему), а `null` – это специально говорим, что объекта нет.

Нужно запомнить, что некоторые "схожие" методы в чистом JS и в DOM API ведут себя по разному, например, когда мы будем искать в массиве что-то, чего там нет – нам вернут `undefined`, а если в DOM – то `null` (это нужно запомнить).



selectors

Селекторы – это специальные выражения, которые позволяют нам решать, подходит элемент по условию или нет.

Условия могут быть разные: например, мы можем искать по тегам или по атрибутам. Сегодня нас будет интересовать только поиск по атрибутам.




data-*

Можно искать по разным атрибутам, но сейчас наиболее распространено использование селекторов по специальным **data**-атрибутам. Что за такие **data**-атрибуты?

Когда мы с вами вкратце говорили про HTML, мы говорили, что у каждого элемента есть атрибуты и все они прописаны в спецификации. Но разработчики спецификации оставили нам возможность для создания собственных атрибутов. Для этого мы должны называть атрибуты с помощью префикса **data-**:

```
8  <body>  
9  |   <h1 data-id="title">Hello, JS!</h1>  
10 |   <script src="js/app.js"></script>  
11 </body>
```



С **data**-атрибутами, как с переменными – мы сами придумываем удобные и имеющие для нас смысл имена.



data-*

Зачем нам создавать свои data-атрибуты, если у элементов уже есть готовые атрибуты? У готовых атрибутов есть своё, чётко прописанное в спецификации предназначение: например, у элемента `script` атрибут `src` отвечает за "расположение" файла с кодом:

```
8  <body>
9    <h1 data-id="title">Hello, JS!</h1>
10   <script src="js/app.js"></script>
11 </body>
```

Поэтому мы вводим свои атрибуты, которые имеют смысл только внутри нашего приложения.



Атрибуты vs Свойства

Важно чётко разграничивать атрибуты и свойства:

- Атрибут – это то, что прописано в разметке
- Свойство – это, что мы получаем через DOM API (свойство объекта)

Браузер делает так, что в большинстве случаев, если вы устанавливаете атрибут, то свойство принимает такое же (или немного модернизированное) значение, и наоборот, устанавливаем свойство – атрибут тоже меняется.

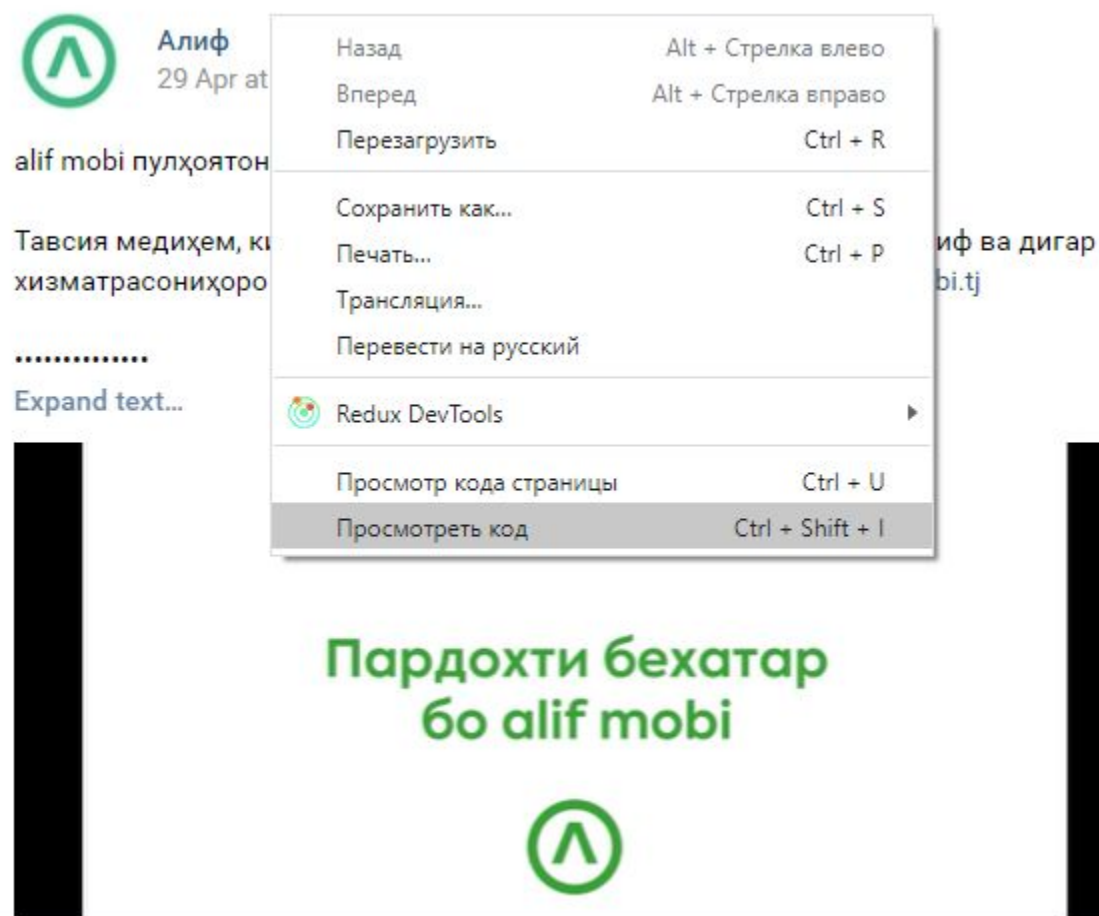
Но, работает это не всегда, поэтому старайтесь использовать свойства.

Кроме того, для некоторых свойств нет атрибутов, например, `textContent` содержит текст, но атрибута у элементов такого нет.



data-*

Если вы зайдёте в Vk и выберите какой-нибудь пост (клик правой кнопкой мыши):



data-*

То увидите, что **data**-атрибуты активно используются:

```
▶<div id="post-149187028_242" class="_post
post page_block all own post--with-likes
closed_comments deep_active" data-post-id=
"-149187028_242" onclick=
"wall.postClick('-149187028_242', event);"
post_view_hash="88e264dc5c808f631a">...</div>
▼<div id="post-149187028_241" class="_post
post page_block all own post--with-likes
closed_comments deep_active" data-post-id=
"-149187028_241" onclick=
"wall.postClick('-149187028_241', event);"
post_view_hash="88e264dc5c808f631a"> == $0
```

Мы научимся работать с ними, а затем уже (на следующих лекциях) рассмотрим остальные.



data-*

```
8 <body>
9   <h1 data-id="title">Hello, JS!</h1>
10  <script src="js/app.js"></script>
11 </body>

> const titleEl = document.querySelector('[data-id="title"]');
< undefined

> titleEl.textContent
< "Hello, JS!"
```



Селектор по **data**-атрибутам выглядит достаточно просто: вы пишете квадратные скобки, имя **data**-атрибута и через равно значение в кавычках (важно, не ставьте нигде лишние пробелы).

Поскольку мы сохранили объект в переменную, то можем спокойно с ним работать.



*EI

```
8 <body>
9   <h1 data-id="title">Hello, JS!</h1>
10  <script src="js/app.js"></script>
11 </body>

> const titleEl = document.querySelector('[data-id="title"]');
< undefined

> titleEl.textContent
< "Hello, JS!"
```

Обратите внимание, мы во всех наших лекциях к переменным, которые предназначены для хранения элементов будем добавлять суффикс **EI** (это просто соглашение, которое позволит понять, для чего нужна переменная).

Вы будете достаточно часто встречаться с таким



Ошибки

При поиске элементов есть два ключевых нюанса:

1. DOM-дерево строится сверху-вниз, поэтому `script` стараются всегда (кроме особых случаев) подключать в самом низу (до `</body>`), чтобы все вышестоящие элементы уже попали в DOM:

```
<body>
  <h1 data-id="title">Hello, JS!</h1>
  <script src="js/app.js"></script>
</body>
```

Если сейчас поменять местами `h1` и `script`, то в скрипте мы не сможем найти никакого `h1`
`document.querySelector` вернёт `null`

2. Вы используете неправильный селектор – обязательно проверяйте свои селекторы в консоли, что вам действительно возвращаются элементы!



null

Вы очень часто будете встречаться с **null** при работе с элементами, здесь всё очень просто:

1. Либо вы ищете не то (используете неправильный селектор)
2. Либо вы ищете не там (не в том месте – об этом чуть позже)
3. Либо вы ищете не тогда, когда нужно (элемент ещё не был создан, либо уже был удалён)

Поэтому каждый раз, когда вы будете встречать ошибку:

```
> document.querySelector('h1').textContent = 'hello';
```

```
✖ ▶ Uncaught TypeError: Cannot set property 'textContent' of null
```

Значит DOM не смог найти то, что вы просили и вернул **null**. А **null** – это не объект и у него нет свойств, методов (и их нельзя ему добавить или прочитать из них).



Prototypes



document.querySelector

Если вы не поленились, распечатали и посмотрели объект `document`, то никакого `querySelector` вы там найти не могли. Но при этом всё работало.

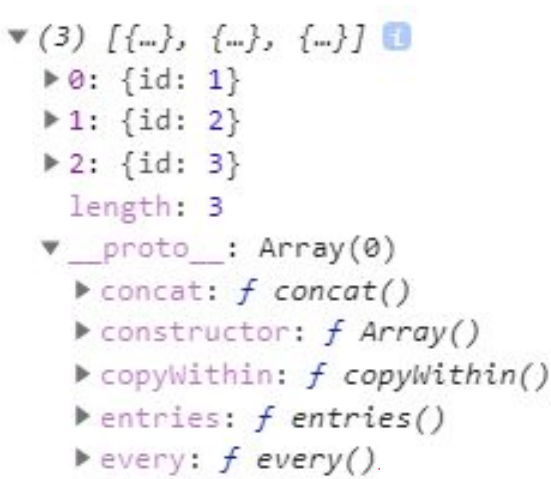
Почему так? Давайте разбираться.



Array

Давайте на время вернёмся к массивам (с ними будет чуть проще) и разберёмся как всё устроено. Итак создадим обычный массив и выведем его в консоль:

```
js > JS app.js > ...  
1  'use strict';  
2  
3  const posts = [  
4    {id: 1, },  
5    {id: 2, },  
6    {id: 3, },  
7  ];  
8  
9  console.log(posts);
```



```
▼ (3) [{...}, {...}, {...}] ⓘ  
  ▶ 0: {id: 1}  
  ▶ 1: {id: 2}  
  ▶ 2: {id: 3}  
  length: 3  
  ▼ __proto__: Array(0)  
    ▶ concat: f concat()  
    ▶ constructor: f Array()  
    ▶ copyWithin: f copyWithin()  
    ▶ entries: f entries()  
    ▶ every: f every()
```

Обратите внимание, у объекта есть свойство `__proto__` или `[[Prototype]]` в новых версиях, в котором хранится другой объект.

Но если вы вспомните задачу с Notifications, то вызывали-то вы метод `every` на самом массиве: `posts.every`? Как же это устроено?



Прототипы

Дело в том, что в JS всё устроено на базе так называемой цепочки прототипов. Если вы что-то спрашиваете у объекта (читаете поле поле), то он сначала смотрит, если у него такое поле или нет.

Если нет, то идёт к объекту, который хранится у него в свойстве `__proto__` (`[[Prototype]]`) и спрашивает его. И так, пока либо свойство не найдено, либо пока в каком-то из `__proto__` (`[[Prototype]]`) не окажется специальное значение `null`, означающее, что объекта дальше нет (и тогда вернётся `undefined`).



Прототипы

Объяснение достаточно сложное, но его очень легко запомнить на следующем примере: представьте, что ребёнку задали в школе нарисовать "пейзаж". Но ребёнок рисовать не умеет. К кому он идёт? Правильно, к родителю (а родитель в JS – это объект, хранящийся в свойстве `__proto__`).

Что делает родитель? Если умеет, то рисует сам, если не умеет – то идёт к своему родителю (и так пока цепочка не прервётся).

Важно: в `__proto__` может быть только один объект. И не путайте `__proto__` с вложенностью в HTML и интерфейсами.



Прототипы

Важно:

- `__proto__` – это про объекты (объект ищет в себе свойство при чтении, если нет – то идёт к объекту, которых хранится в `__proto__`)
- интерфейсы – это про то, какие в объекте (и во всей цепочке `__proto__`) должны быть свойства
- вложенность – это про то, какой элемент (внутри DOM-дерева) какие элементы содержит



Прототипы



Прототипы

То же самое произошло с `document.querySelector`. В самом объекте `document` его (этого метода) нет, зато он есть в `__proto__` его `__proto__`.



Прототипы

Всё, что попадает в `__proto__` нашего объекта описано в документации MDN как

```
Array.prototype.concat()
```

```
Array.prototype.copyWithin()
```

```
Array.prototype.entries()
```

```
Array.prototype.every()
```

```
Array.prototype.fill()
```

```
Array.prototype.filter()
```

```
Array.prototype.find()
```

И на самом деле, у всех массивов в свойстве `__proto__` будет находиться один и тот же объект (вспомните аналогию про детей – у одного родителя может быть много детей).



Свойства и атрибуты



Свойства и атрибуты

Когда мы пишем HTML-документ и создаём там элементы, то мы пишем атрибуты:

```
<body>  
    
  <h1 data-id="title">Hello, JS!</h1>  
  <script src="js/app.js"></script>  
</body>
```

Когда мы пытаемся получить к ним доступ, мы делаем это через свойства объектов или через метод `getAttribute()`. Имена свойств почти всегда совпадают с именем атрибута (есть ряд исключений, но о них чуть позже).

Установить значение можно либо через свойства, либо через метод `setAttribute()`.

Напоминаем, что атрибут это выражение `name="value"` внутри открывающего тега.

Если у атрибута пустое значение, то он может писаться как просто `name`.



Свойства и атрибуты

```
<body>
  
  <h1 data-id="title">Hello, JS!</h1>
  <script src="js/app.js"></script>
</body>
```

Важно, чтобы вы понимали разницу между доступом через свойство и доступом через метод `getAttribute()`:

```
> const logoEl = document.querySelector('[data-id="logo"]');
< undefined
> logoEl.src
< "http://127.0.0.1:5500/img/logo_js.svg"
> logoEl.getAttribute('src');
< "img/logo_js.svg"
```

Через `getAttribute()` мы получаем доступ к тому, как оно "было записано" в HTML. А через свойство – к текущему значению (т.е. браузер переделал ссылку и подставил полный адрес). В большинстве случаев нам нужно работать именно через свойства.



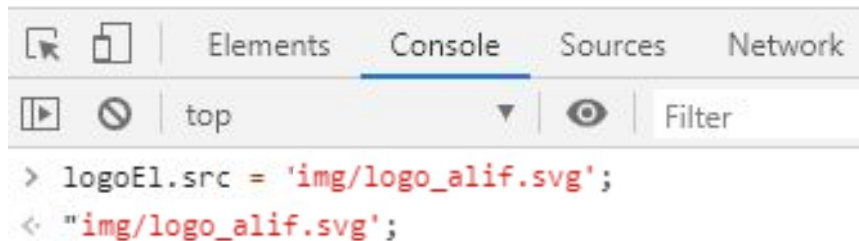
Свойства и атрибуты

Вся прелесть DOM API заключается в том, что вам достаточно изменить свойство, всё остальное браузер сделает сам, например:



Hello, JS!

Мы просто поменяли ссылку, а браузер сходил по этой ссылке, скачал картинку и подставил вместо нашей картинки



Обратите внимание, адрес изображения указывается относительно html-страницы, а не относительно скрипта.



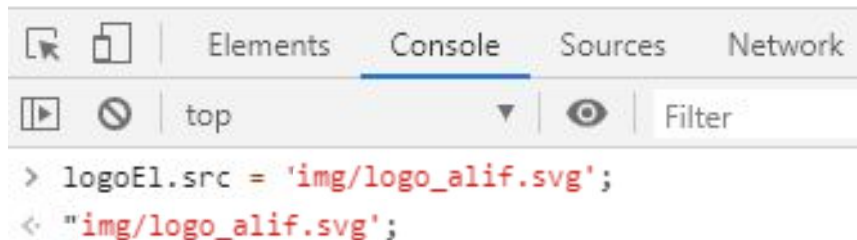
Свойства и атрибуты

При этом установка свойства может производиться "удобным" способом:



Hello, JS!

Т.е. не обязательно указывать полный адрес, браузер всё сделает сам



При этом работая со свойствами – мы можем работать не только со строками, в то время как `setAttribute` работает только со строками (эту деталь мы закрепим на следующих занятиях).



Elements



Elements

Вкладка **Elements** – в DevTools ваш лучший друг при работе с DOM. В ней вы можете не изменяя самого HTML, редактировать элементы, их атрибуты, текст и т.

Д.:

```
<!DOCTYPE html>
<html lang="en">
  <head>...</head>
  <body>      ДВОЙНОЙ КЛИК ЛЕВОЙ КНОПКОЙ МЫШИ
    
...  <h1 data-id="title">Hello, Alif Skills!</h1> == $0
    <script src="js/app.js"></script>
    <!-- Code injected by live-server -->
    <script>...</script>
  </body>
</html>
```

Естественно, все ваши изменения пропадут после обновления страницы (они не сохраняются в ваших файлах).



Elements

```
<!DOCTYPE html>
<html lang="en">
  <head>...</head>
  <body>
    ...  == $0
    <h1 data-id="title">Alif!</h1>
    <script src="js/main.js"></script>
    <!-- Code injected by live-server -->
    <script>...</script>
  </body>
</html>
```

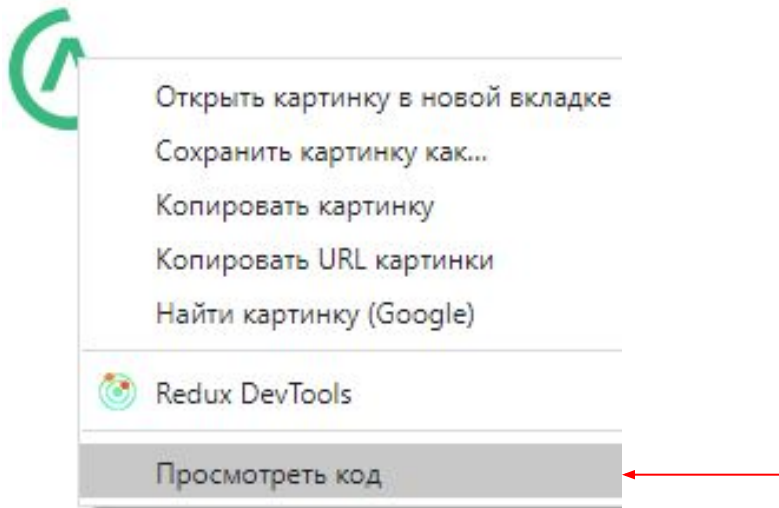
клик правой кнопкой мыши

- Add attribute
- Edit attribute
- Edit as HTML
- Duplicate element
- Delete element
- Cut
- Copy
- Paste
- Hide element
- Force state
- Break on
- Expand recursively
- Collapse children
- Capture node screenshot
- Scroll into view
- Focus
- Badge settings...
- Store as global variable

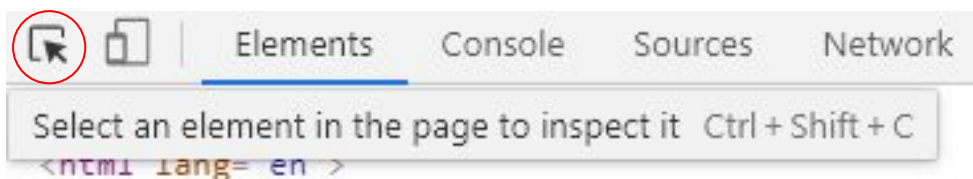


Elements

Сопоставить элемент на странице объекту в панельке Elements вы можете либо щёлкнув на элементе правой кнопкой мыши и выбрав "Посмотреть код":



Либо в самой панельке выбрать инструмент выделения и кликнув левой кнопкой мыши на любой элемент на странице:



setTimeout



setTimeout

По-отдельности мы научились всё менять, осталось научиться это делать из скрипта. Кроме того, нужно обеспечить смену данных только через 2 секунды.

Для запуска функции через какое-то время у нас есть встроенная функция `setTimeout*`:

```
1  'use strict';  
2  
3  setTimeout(() => {  
4    // Наш код  
5  }, 2000);
```

Первый параметр – функция, которую нужно выполнить (у нас – стрелочная функция), второй параметр – время в миллисекундах, через которое надо выполнить.

Примечание*: мы специально не даём прямо в лекциях ссылки на страницы MDN (учитесь гуглить "MDN setTimeout" – первая же ссылка в Google будет вашей).



setTimeout

Важно: на собеседованиях спрашивают, точно ли через две секунды выполнится функция?

Правильный ответ – не точно. Функция выполнится не раньше, чем через две секунды. Почему не раньше? Ответ на этот вопрос вы узнаете из следующей лекции.



setTimeout

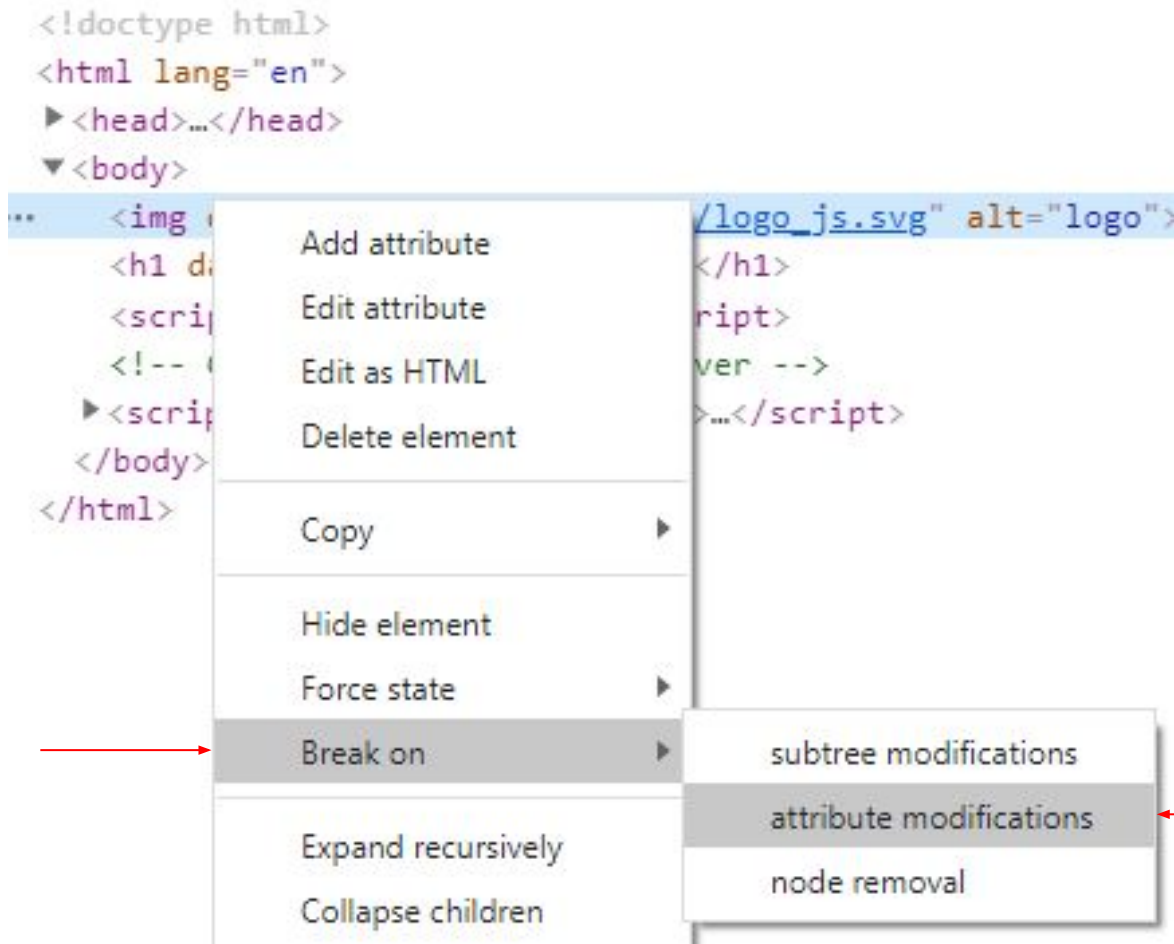
Это решение рабочее, но не совсем правильное – предпочитают элементы не искать внутри функций, а искать их снаружи и сохранять в переменную, но до следующей лекции нас это устроит:

```
setTimeout(() => {  
  const logoEl = document.querySelector('[data-id="logo"]');  
  const titleEl = document.querySelector('[data-id="title"]');  
  
  logoEl.src = 'img/logo_alif.svg';  
  titleEl.textContent = 'Hello, Alif Skills!';  
}, 2000);
```



Elements

У вкладки Elements есть удобная функция: вы можете "остановить" код в дебаггере при изменении атрибута элемента:



Elements

Это вам поможет отловить, кто же именно и где меняет атрибуты вашего элемента:

```
1 'use strict';
2
3 setTimeout(() => {
4     const logoEl = document.querySelector('[data-id="logo"]');
5     const titleEl = document.querySelector('[data-id="title"]');
6
7     logoEl.src = 'img/logo_alif.svg';
8     titleEl.textContent = 'Hello, Alif Skills!';
9 }, 2000);
10
```



Итоги



Итоги

В этой лекции мы обсудили работу с DOM. Понимать DOM важно, поскольку какие бы фреймворки вы не использовали, вам рано или поздно придётся "опуститься" до уровня DOM.

Мы не рассматриваем в этом курсе всех тонкостей: часть из них будет рассматриваться позже, а другая часть вам будет не нужна, в связи с тем, что вы будете использовать React.

Мы снова рекомендуем не терять времени зря и начать приучать себя читать спецификации и ознакомиться со страничкой [Web API](#) на MDN.



Домашнее задание



Орг.моменты

Практикум состоит из 8 обязательных занятий. Мы выкладываем новые занятия каждый понедельник в 14:00 (по Душанбе), кроме первой недели.

Каждое воскресенье в 23:59 (по Душанбе) дедлайн сдачи домашнего задания. Дедлайн – это предельный срок, до которого вы должны сдать ДЗ.

Если не успеете сдать в срок домашнее задание, тогда этот практикум будет для вас закончен и вы сможете зарегистрироваться на запуск следующего через несколько месяцев.

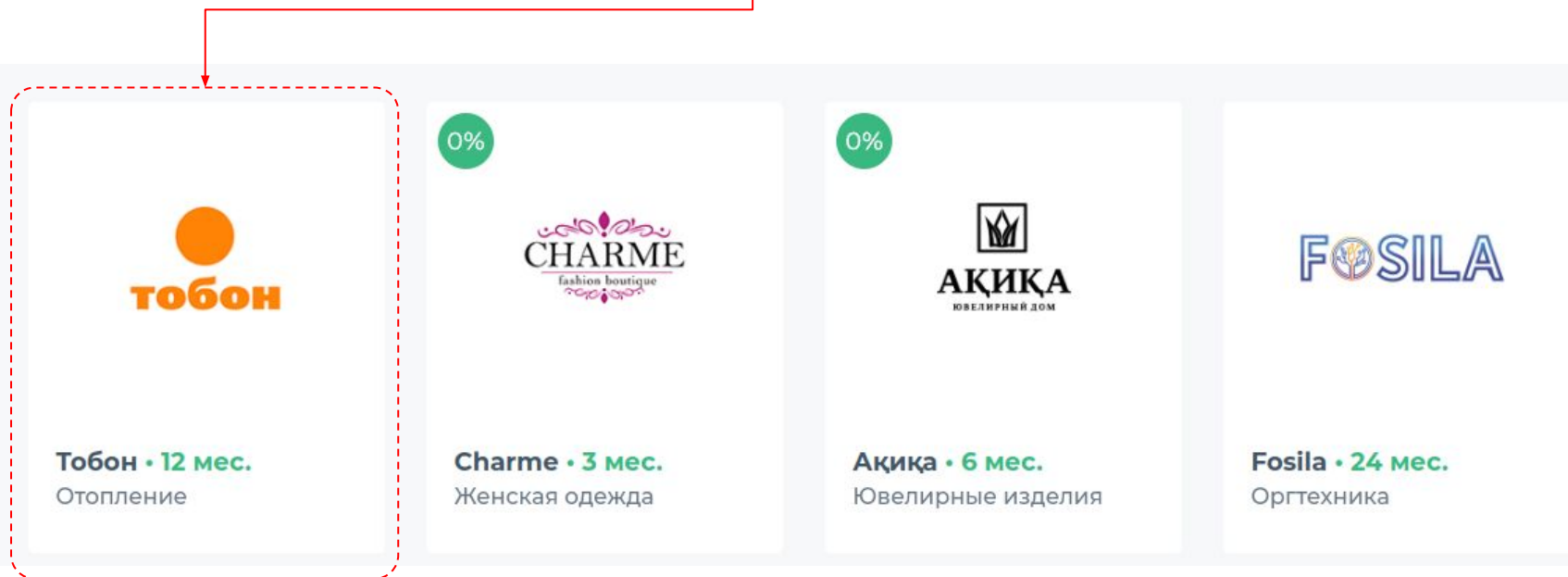
Все вопросы вы сможете задавать в [Телеграм канале](#).



ДЗ №1: Tobon

Надеюсь вы помните ДЗ с Salom и ваш любимый объект с [id=tobon](#).

Так вот пришла пора сделать эту карточку (пока без оформления):



ДЗ №1: Tobon

Вам дана следующая разметка:

```
<body>
  <div data-block="partner">
    <a data-id="partner-link" href="...">
      
      <div>
        <span data-id="title">...</span>&nbsp;&bull;&nbsp;&nbsp;<span data-id="period">...</span>
      </div>
      <div data-id="category">...</div>
    </a>
  </div>
  <script src="js/app.js"></script>
</body>
```

Там, где стоят ... вы должны средствами JS вписать нужные значения.



ДЗ №1: Tobon

Как это сделать? Вы делаете функцию, которую называете `bindPartnerToEl`, которая на вход принимает объект со свойствами и элемент, к которому этот объект нужно привязать:

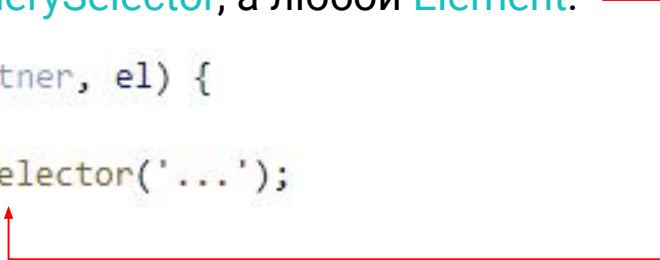
```
1  'use strict';
2
3  function bindPartnerToEl(partner, el) {
4      // ваш код
5      const linkEl = el.querySelector('...');
6  }
7
8  const partner = {
9
10
11      Ваши данные
12
13
14
15  };
16
17  const partnerEl = document.querySelector('...');
18  bindPartnerToEl(partner, partnerEl);
```



ДЗ №1: Tobon

И дальше выясняется забавная вещь: оказывается, не только `document` может искать внутри себя через `querySelector`, а любой `Element`:

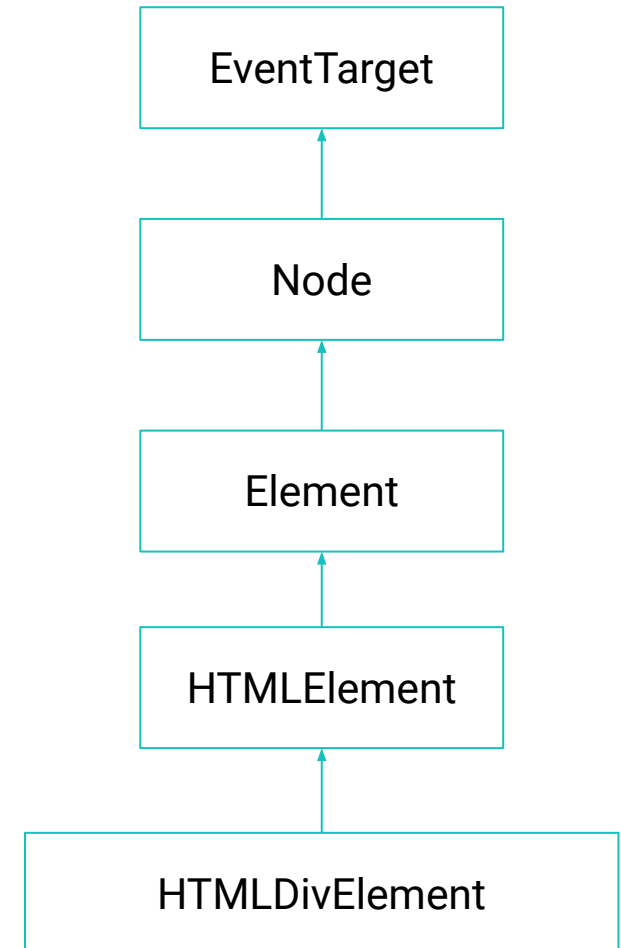
```
function bindPartnerToEl(partner, el) {  
  // ваш код  
  const linkEl = el.querySelector('...');  
}
```



ДЗ №1: Tobon

На всякий случай напоминаем вам иерархию интерфейсов (иерархия интерфейсов означает, что все методы, которые есть в `Element` есть и в `HTMLDivElement`, которым является `div` с `data-block="partner"`):

Важно: иерархия интерфейсов ничего не говорит о том, как это организовано "внутри" (а именно, что должно храниться в `__proto__`, а что в самом объекте).



ДЗ №1: Tobon

Как бот будет проверять задачу: бот будет программно менять объект и сам элемент (подставляя другой), после чего вызывать вашу функцию и сверять результат.

Каталог в архиве должен называться [tobon](#).



ДЗ №2: ID Владельца

Вам нужно написать функцию, которая при вызове находит элемент с заданным `data-id` и вытаскивает оттуда значение `"data-ownerid"` (идентификатор владельца):

```
<div data-id="1" data-type="post" data-ownerid="24234">  
  Первый пост в нашей социальной сети!  
</div>
```

Ваша функция должна называться `extractOwnerId` и принимать на вход один параметр – `postId`. Возвращать она должна значение атрибута `data-ownerid`:

```
js > JS app.js > ...  
1   'use strict';  
2  
3   function extractOwnerId(postId) {  
4     const selector = `[data-id="${postId}"]`;   
5     // ваш код  
6     return ownerId;  
7   }
```



ДЗ №2: ID Владельца

Для того, чтобы решить эту задачу, нам нужно разобрать вот эту конструкцию:

```
const selector = `[data-id="${postId}"]`;
```

Она называется template literals. Что это значит, это значит, что на место `${postId}` просто подставляется значение `postId` и всё превращается в строку (смотрим в консоли):

```
> const postId = 100;  
< undefined  
  
> `data-id=${postId}`  
< "data-id=100"
```

``` - называют обратными кавычками или backticks. Такая подстановка работает только внутри них (внутри одинарных или двойных кавычек работать не будет).



# ДЗ №2: ID Владельца

Кроме того, мы с вами говорили, что почти все атрибуты отображаются в свойства "как есть". Так вот, `data-*` атрибуты не такие.

Вам нужно провести небольшое исследование и посмотреть, в какие свойства они отображаются в объекте. Заглянуть надо в свойство `dataset`.



# ДЗ №2: ID Владельца


Как бот будет проверять задачу: бот будет программно менять `data-id` и `data-ownerid`, после чего вызывать вашу функцию и сверять результат.

Каталог в архиве должен называться `ownerid`.



# ДЗ №3: Loader

Все вы видели специальный виджет, показывающий вам, что страница что-то делает и нужно подождать. Выглядит это обычно, следующим образом:

 Загружаем статистику

На самом деле, не факт, что в этот момент действительно что-то происходит. Некоторые хитрые веб-приложения просто пользователю показывают "loader" (вот эту картинку загрузки), чтобы он не переживал и зазря не обновлял страницу.



# ДЗ №3: Loader

Показывать и скрывать этот элемент можно многими способами. Мы рассмотрим один из них. Есть такое CSS-свойство, называется `display`. Нас будут интересовать два значения:

1. `block` – элемент показывается (см. level 0 для подробностей)
2. `none` – элемент не показывается (браузер рисует страницу так, как будто этого элемента вообще нет)



# ДЗ №3: Loader

Вам дана разметка (картинку, конечно было бы лучше задать фоном, но оставим пока так):

```
1 <!DOCTYPE html>
2 <html lang="ru">
3 <head>
4 <meta charset="UTF-8">
5 <meta name="viewport" content="width=device-width, initial-scale=1.0">
6 <title>Document</title>
7 <link rel="stylesheet" href="css/style.css">
8 </head>
9 <body>
10 <div data-id="loader">
11 Пожалуйста, подождите
12 </div>
13 <script src="js/app.js"></script>
14 </body>
15 </html>
```

Картинку вы можете сгенерировать сами на сайтах вроде <https://loading.io> (главное – сохраните её под именем `loader.gif`, бот будет искать картинку именно с таким именем).





# ДЗ №3: Loader

Что нужно сделать: нужно в `app.js` написать код, который через 5 секунд после загрузки страницы скрывает этот элемент, выставляя нужное css-свойство. Как это сделать? Для доступа к CSS есть свойство `style` (работает со всеми элементами, не только с `body`):

```
> document.body.style
< ▼ CSSStyleDeclaration {alignContent: "", alignItems: "", alignSelf: "", alignmentBaseline: "", all: "", ...}
 alignContent: ""
 alignItems: ""
 alignSelf: ""
 alignmentBaseline: ""
 all: ""
 animation: ""
 ...
 background: ""
 backgroundAttachment: ""
 backgroundBlendMode: ""
 backgroundClip: ""
 backgroundColor: ""
```

Названия у них почти так же, как в самом CSS, только дефисы заменены на имена, валидные для JS: `background-color` пишется как `backgroundColor`.



# ДЗ №3: Loader

Соответственно, вам нужно поэкспериментировать с этим свойством (с учётом того, что мы говорили раньше) и добиться того, чтобы ваш элемент скрывался.

Бот будет проверять, что при загрузке страницы элемент видим, а через 5 секунд – скрыт.

Каталог в архиве должен называться [loader](#).



# ДЗ №4: Thanos Effect

Всем вам понравилось ДЗ про Таноса. Неплохо бы не просто в консольку выводить отфильтрованные результаты, а и научиться их скрывать со страницы? Вот ваша разметка:

```
9 <body>
10 <div data-id="posts">
11 <div data-type="post" data-id="5">
12 Пятый пост!
13 </div>
14 <div data-type="post" data-id="4">
15 Четвёртый пост!
16 </div>
17 <div data-type="post" data-id="3">
18 Третий пост!
19 </div>
20 <div data-type="post" data-id="2">
21 Второй пост!
22 </div>
23 <div data-type="post" data-id="1">
24 Первый пост!
25 </div>
26 </div>
27 <script src="./js/app.js"></script>
```



# ДЗ №4: Thanos Effect

Что нужно сделать: написать функцию `thanosEffect`, которая принимает на вход элемент, внутри которого и будет производиться "очистка":

```
1 'use strict';
2
3 function thanosEffect(el) {
4 const list = el.querySelectorAll('...');
5 // Ваш код
6 }
7
8 const postsEl = document.querySelector('[data-id="posts"]');
9 thanosEffect(postsEl);
```

Но если вы попытаете вызвать на `list` любой из методов `Array`, то ничего у вас не получится по одной простой причине: то, что возвращает `querySelectorAll` не содержит нигде в цепочке прототипов `Array`. Поэтому придётся вам почитать документацию на `Array` и найти там функцию, которая позволяет преобразовать "любой итерируемый объект" в массив (ищите наверху).



# ДЗ №4: Thanos Effect

Как скрывать? Когда мы ставим `display: none` через CSS элемент перестаёт занимать место на странице (и всё сдвигается вверх):

|                 |   |                 |
|-----------------|---|-----------------|
| Пятый пост!     |   | Четвёртый пост! |
| Четвёртый пост! |   | Второй пост!    |
| Третий пост!    | → |                 |
| Второй пост!    |   |                 |
| Первый пост!    |   |                 |

Но в гугле ведь ничего не сдвигалось? Чтобы добиться такого эффекта, мы будем использовать другое CSS-свойство, оно называется `visibility`. Если ему поставить значение `hidden`, то получится вот так:

|                 |   |                 |
|-----------------|---|-----------------|
| Пятый пост!     |   | Четвёртый пост! |
| Четвёртый пост! |   |                 |
| Третий пост!    | → | Второй пост!    |
| Второй пост!    |   |                 |
| Первый пост!    |   |                 |



## ДЗ №4: Thanos Effect

Как вы уже поняли, нужно убрать посты с чётными индексами (включая нулевой). Индексы считаются по тому, как посты встречаются в DOM. Чтобы решить эту задачу, вам нужно после преобразования в массив воспользоваться последовательно двумя функциями `filter` и `forEach`:

```
items.filter(...).forEach(...);
```

Как это работает? Метод `filter` возвращает новый массив, в который попадают только те, кто прошёл фильтр. И уже на этом массиве вызывается метод `forEach`, в котором вызывается функция для каждого элемента из этого массива (в ней, в этой функции, вы и должны выставлять стили).

Каталог в архиве должен называться `thanos-effect`.



# ДЗ №5: VK

Давным-давно, главная страница сервиса Vk выглядела вот так:



The screenshot shows the old VK.com homepage. At the top is a blue header with the VK logo and the word "контакте" in white. To the right of the header are two buttons: "вход" (login) and "регистрация" (registration). Below the header, on the left, is a login form with two input fields: "E-mail или Логин:" and "Пароль:". Below these fields are two buttons: "Вход" and "Регистрация". To the right of the login form is a yellow banner with the text "Добро пожаловать" (Welcome). Below the banner, the main content area has a heading "ВКонтакте - универсальное средство поиска знакомых." (VKontakte - a universal means of finding acquaintances). Below this heading is a paragraph: "Мы хотим, чтобы друзья, однокурсники, одноклассники, соседи и коллеги всегда оставались в контакте." (We want friends, classmates, schoolmates, neighbors and colleagues to always stay in contact). Below this paragraph is a line of text: "Нас уже 50 000 000." (We already have 50,000,000). A red arrow points to the number "50 000 000". Below this line is another heading: "ВКонтакте - самый посещаемый сайт в России и Украине." (VKontakte - the most visited site in Russia and Ukraine). Below this heading is a paragraph: "С помощью этого сайта Вы можете:" (With the help of this site you can:). Below this paragraph is a list of three bullet points: "■ Найти людей, с которыми Вы когда-либо учились, работали или отдыхали." (Find people with whom you have ever studied, worked or rested), "■ Узнать больше о людях, которые Вас окружают, и найти новых друзей." (Find out more about people who surround you and find new friends), and "■ Всегда оставаться в контакте с теми, кто Вам дорог." (Always stay in contact with those who are dear to you). Below the list of bullet points are two buttons: "Вход" and "Регистрация". At the bottom of the page is a footer with the text: "В Контакте © 2006-2009 русский украинська беларуская (тарашкевіца) English српски magyar Azərbaycan все языки »" (VKontakte © 2006-2009 Russian Ukrainian Belarusian (Taraškievica) English Serbian Magyar Azerbaijani all languages »).

Что интересно, подчёркнутое число генерировалось динамически на JS. Самое забавное, что оно не считало кол-во зарегистрировавшихся. А просто с определённым интервалом времени добавляла случайное число 😊.



# ДЗ №5: VK

У нас есть функция `setInterval`, которая позволяет с определённым интервалом в миллисекундах выполнять указанную вами функцию (и если `setTimeout` запускала функцию один раз, то `setInterval` – пока не остановим):

```
js > JS app.js > ...
1 'use strict';
2
3 setInterval(() => {
4 // ваш код
5 }, 1000);
```

Ваша разметка:

```
<body>
 <div>Нас уже 0</div>
 <script src="js/app.js"></script>
</body>
```

Ваша задача: каждую секунду увеличивать значение на 10.





## ДЗ №5: VK

Для этого не нужно создавать дополнительных переменных.

Вспомните о том, что элементы это тоже объекты, и в них точно так же можно создавать свойства (это считается плохой идеей – но наша задача вас познакомить с этим способом, т.к. именно так работают многие фреймворки).

Создайте в элементе с `data-id="counter"` свойство `__counterValue` (обратите внимание на два подчёркивания спереди – оно нужно, чтобы мы случайно не затёрли стандартные свойства, так обычно делают создатели библиотек, например, React), в которое записывайте текущее значение.

Получать сам элемент можно в `setInterval` (через `document.querySelector`). В следующей лекции мы поговорим, почему это не лучшее решение.



# ДЗ №5: VK

Каталог в архиве должен называться [vk](#).



Спасибо за внимание

**alif skills**

2023г.

